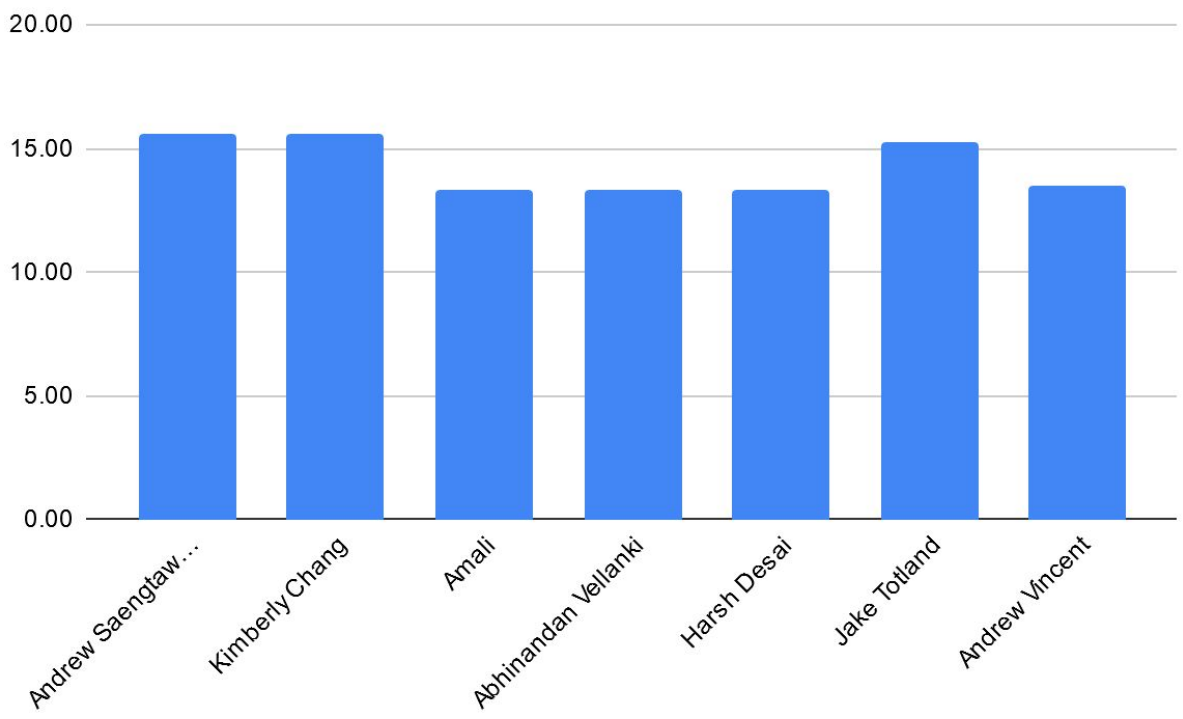
Automated Supermarket Report 2

Group 4
github.com/as2580/SoftwareEngineeringGroup4

Table of Contents

Section 0: Individual Contributions Breakdown

		Andrew Saengtawesin	Kimberly Chang	Amali Delauney	Abhinandan Vellanki	Harsh Desai	Jake Totland	Andrew Vincent
Section 1: Interaction Diagrams	30			33.3%	33.3%	33.3%		
Section 2: Class Diagrams and Interface Specification	10			33.3%	33.3%	33.3%		
Section 3: System Architecture and Design	15	50.0%	50.0%					
Section 4: Algorithms and Data Structures	4						50.0%	50.0%
Section 5: User Interface and Implementation	11						50.0%	50.0%
Section 6: Design of Tests	12						50.0%	50.0%
Section 7: Project Management	18	45.0%	45.0%				10.0%	
Total	100	15.60	15.60	13.33	13.33	13.33	15.30	13.50



Section 0-1

Section 1: Interaction Diagrams

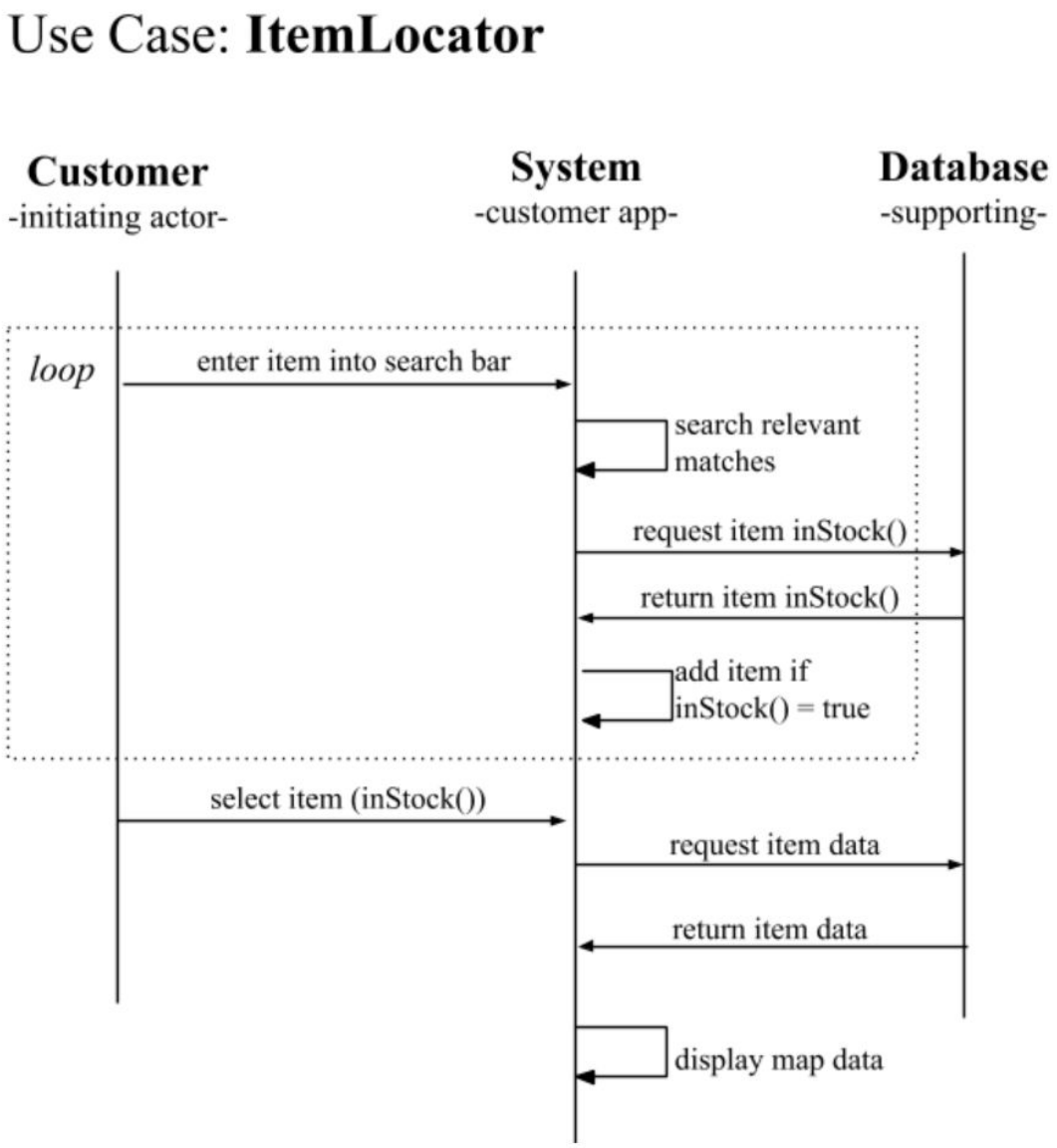


Figure 1-1

- **System** has a high coupling because it has to communicate to the database and the customer.
- **Database** is an Expert Doer because it has knowledge if an item inStock and all the information on an item.
- **Customers** have low coupling because the system makes it easy for customers to communicate.

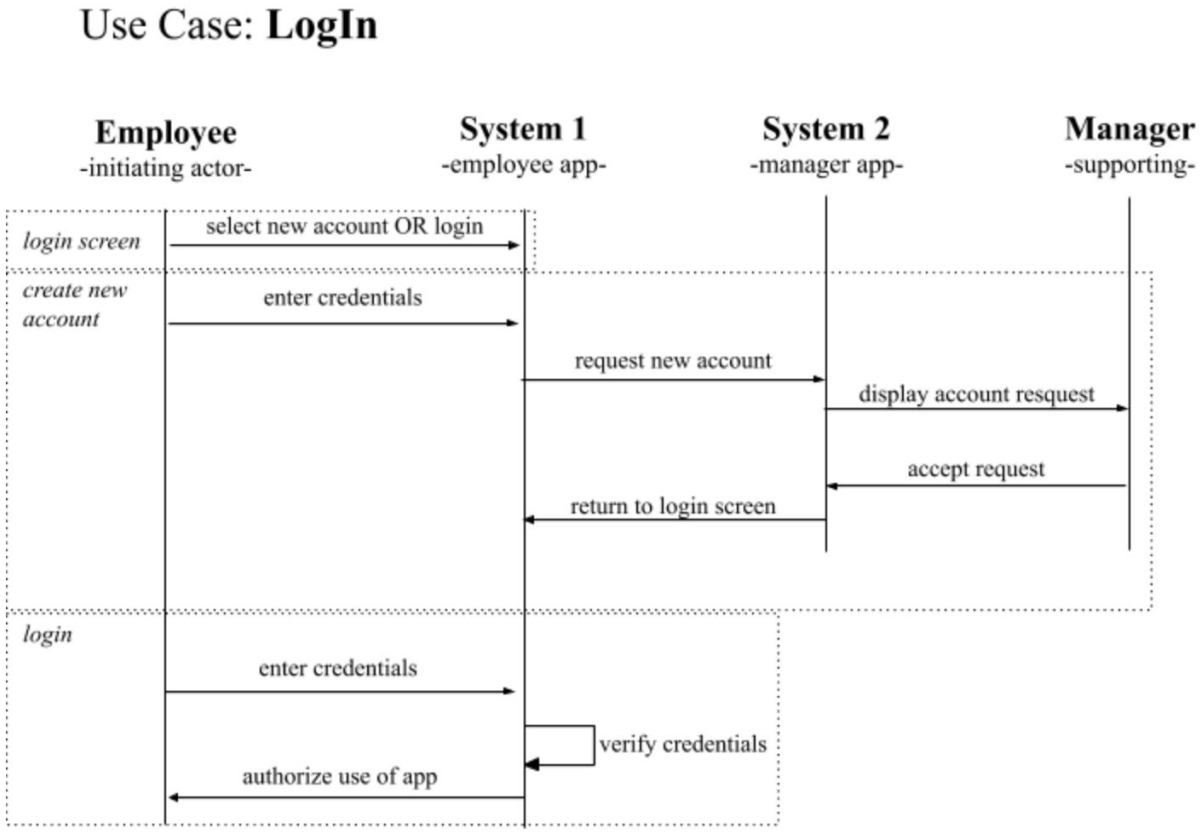


Figure 1-2

- **Employees** have high cohesion because they only have to type their correct credentials. (Single Responsibility Principle)
- **System 1** has low coupling because it only has to communicate with system 2.
- **System 2** has low coupling because the system does not have to send a lot of messages because system 2 shares some responsibility with system 1.

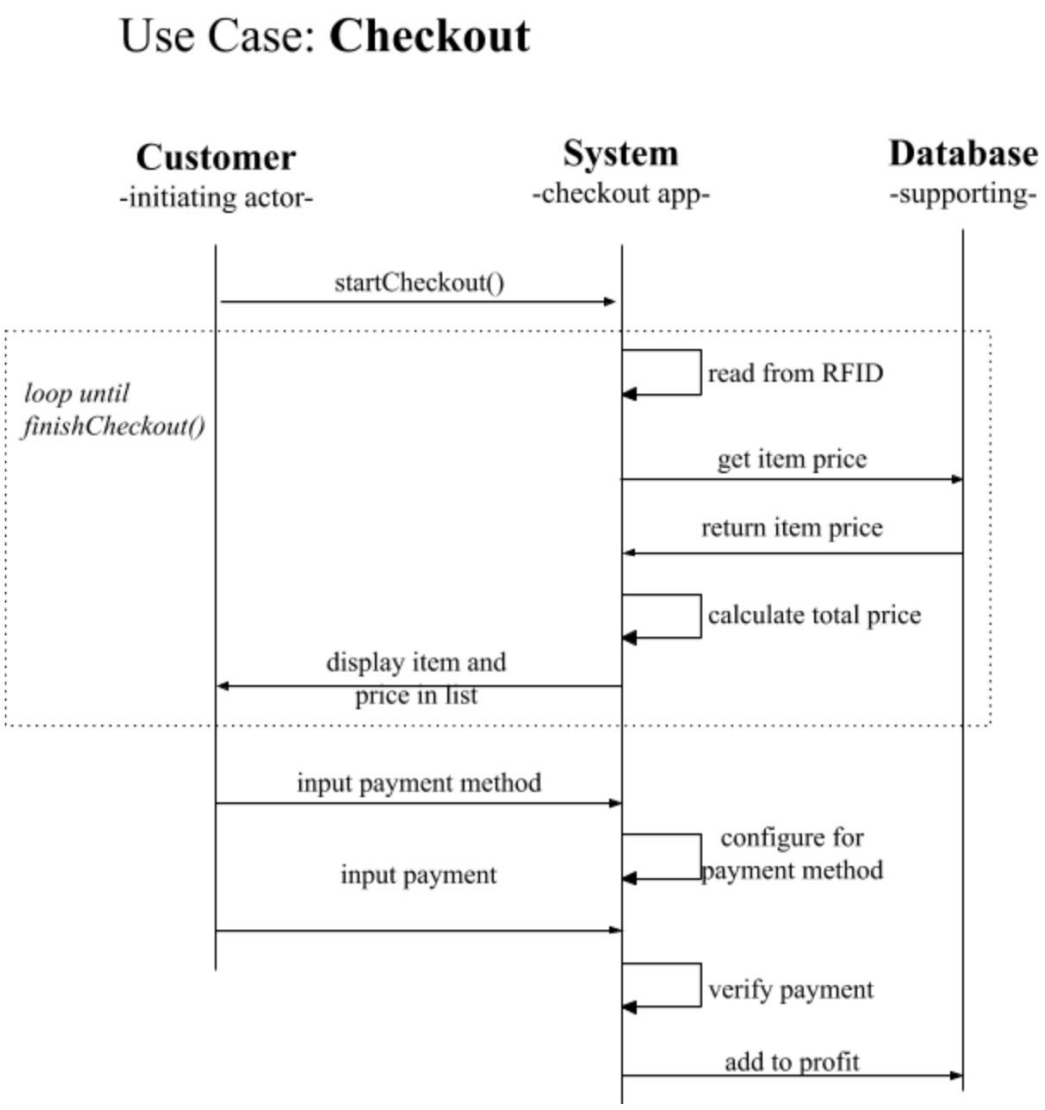


Figure 1-3

- **System** has a low cohesion as it performs many calculations
- **Database** is an Expert Doer because it has full knowledge of every item in the store, including it's price which it delivers to the system

Use Case: **ReturnItem**

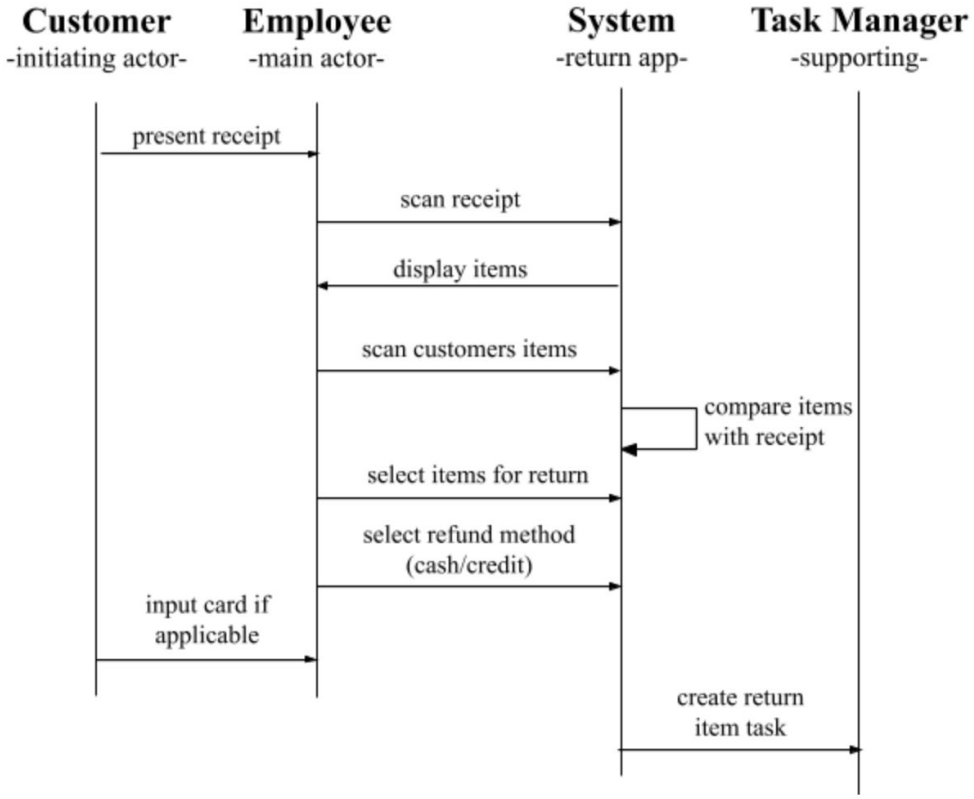


Figure 1-4

- **Customers** have low coupling because the system makes it easy for them to communicate.
- **Employees** have high coupling because they have to constantly communicate with the system.
- **System** has high cohesion because it only has to work with the items on the receipt and create one task

Use Case: **PriceChecker**

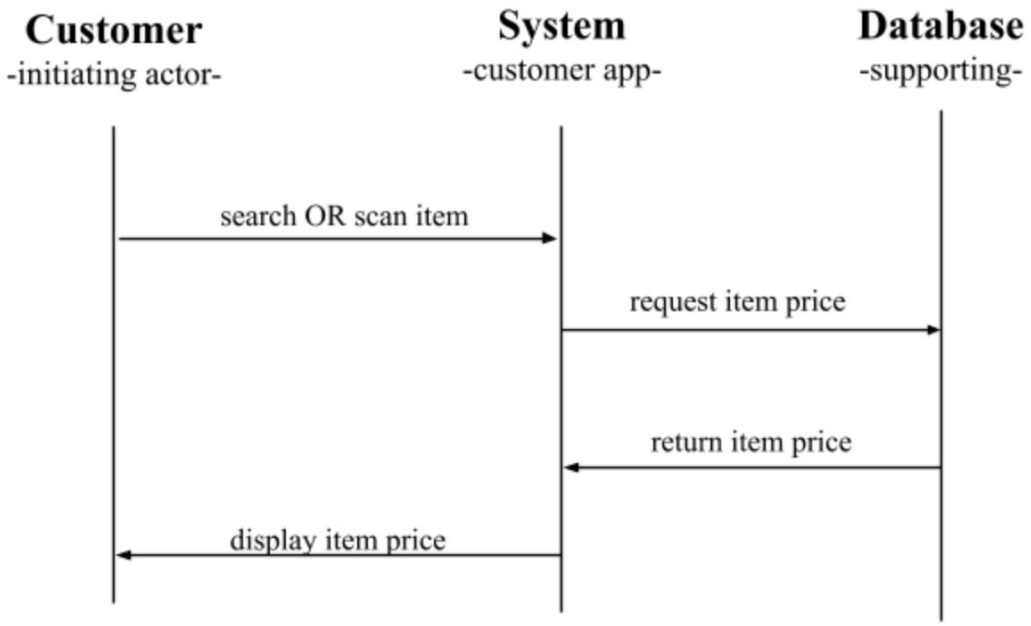


Figure 1-5

- **Each object** has high cohesion as they have only one responsibility.
- **Database** is an Expert Doer because it has full knowledge of every item in the store, including its price which it delivers to the system

Section 2: Class Diagram and Interface Specification

Section 2.1: Class Diagram

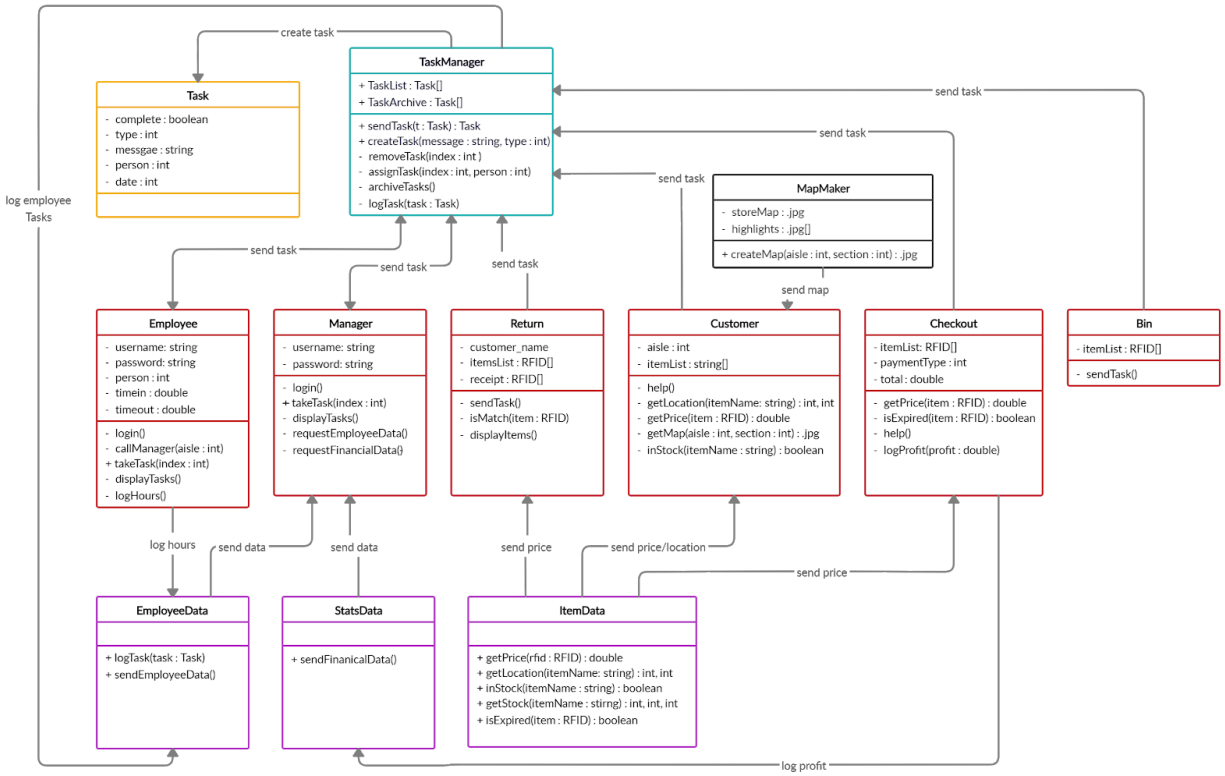


Figure 2.1-1

Section 2.2: Data Types and Operation Signatures

Class: Task **// structure, holds info about task**

```
boolean complete        // true = task completed
int type                // 1 = manager, 0 = employee
string message          // message to be displayed
int person              // identifies which person is completing a task using
                           employeeID (or 000 for manager)
int date                // date in mmddyy form as an integer
```

Class: Task Manager

```
Task[] TaskList         // list of all Tasks
Task[] TaskArchive      // list of every Task, updated at the end of each day

sendTask(Task t)        // sends a task to a controller type (employee or manager)
createTask(int type, string message)
        // receives type and message info from one of the apps controllers
        // creates Task based on info and adds to TaskList
        // calls sendTask(Task t, int type) to send task to a different controller
removeTask(int index)   // removes a task at this index of the array
assignTask(int index, int person)   // assigns an indexed Task to a specific purpose
archiveTasks()          // adds tasks to TaskArchive upon end of day or completed
logTask(Task task)      // adds completed task to employee database
```

Class: MapMaker

```
.jpg storeMap           // map image of store
.jpg highlights         // red highlight images to be overlayed on storeMap

createMap(int aisle, int section)   // returns an image of store with aisle and section
                                       highlighted for clarity
```

Class: Employee

```
string username         // username of employee for login
string password         // password of employee for login
int person              // employeeID to assign to Task
double timeIn           // time employee logs in
double timeOut          // time employee logs out

login()                 // grants access to app functionality
```

callManager(int aisle) // sends Task for manager help at a specific aisle
takeTask(int index) // assigns Task at index to this employee
displayTasks() // display all tasks on screen with buttons to takeTask()
logHours() // sends total hours worked data to Class EmployeeData

Class: Manager
string username // username of manager for login
string password // password of manager for login
login() // grants access to app functionality
takeTask(int index) // assigns Task at index to the manager
displayTasks() // display all tasks on screen with buttons to takeTask()
requestEmployeeData() // requests and displays detailed table of employee data
requestFinancialData() // requests and displays detailed table of financial data

Class: Return
int transactionID // ID of the transaction (for identification)
int transactionDate // date of the transaction (for identification)
RFID[] itemsList // array of all items scanned using RFID scanner
RFID[] receiptList // array of all items purchased as shown on receipt

sendTask() // sends Task for an item to be reshelved
isMatch(RFID item) // returns true if scanned item matches RFID on receipt
displayItems() // displays all scanned items on screen

Class: Customer
int aisle // isle number of this customer terminal
string[] itemList // list of all item names for customer to search through

help() // sends Task to employee of customer in need of assistance
getLocation(string itemName) // requests and returns aisle# and section# of item
getPrice(RFID rfid) // requests and returns price of scanned item
getMap(int aisle, int section) // requests and returns a map detailing the location of item
inStock(string itemName) // returns true if item is in stock

Class: Checkout
RFID[] itemList // list of items scanned by RFID scanner
int paymentType // payment type either cash (0) or credit (1)
double total // total price of items to be purchased

Class: Bin

RFID[] itemList	// list of items scanned by RFID scanner
sendTask()	// sends task for an item to be reshelved

Class: EmployeeData

logTask(Task task)	// adds completed task to employee database
sendEmployeeData()	// sends complete table of employee data

Class: StataData

sendFinancialData()	// sends complete table of financial data

Class: ItemData

getPrice(RFID rfid)	// returns price of scanned item
getLocation(string itemName)	// returns aisle# and section# of item
inStock(string itemName)	// returns true if item is in stock
getStock(string itemName)	// returns number stock in total, shelved, and in storage
isExpired(RFID rfid)	// returns true if item is past sell by date

Section 2.3: Traceability Matrix

Domain Concepts	**Software Classes**										
	StatsData	ItemData	Task	Task Manager	Return	Checkout	Customer	Manager	Employee	Bin	MapMarker
Price Holder		X				X	X				
Stock Manager		X						X			
Item Locator		X					X				
Map Maker											X
Employee Database									X		
Financial Database	X							X			
Task Manager			X	X	X				X	X	
Teremial				X	X	X					X

Section 3: System Architecture and System Design

Section 3.1: Architectural Styles

This system is a combination of the Backboard System, Database-Centric Architecture, and Peer-to-Peer. The system is mostly the former two styles because communication typically occurs between one application and the database. The Customer Application, Employee Application, Manager Application, Checkout Terminal application, and Returns Terminal application do not generally directly interact with each other. Rather, they usually query a database to obtain the necessary information. More time-sensitive events are done with a Peer-to-Peer style. Generally, Peer-to-Peer is used for immediate alerts.

The Customer Application typically queries the database. Item search and price checking queries the database to obtain information about the price and/or location of the item. For customers logged in on their smart device, the customers would also query the database for information on items to be added to their shopping list; information about items on their shopping list may be saved on the user's device to allow the user to check their shopping list even when offline. At a Customer Assistance Terminal, the Customer Application will ping the Employee Application with an immediate notification; this will cause the Employee Application to immediately query the database for an updated task board and give an alert to employees that a new task is available. The Checkout Terminal application and Returns Terminal application will similarly ping the Employee Application when a Customer requires assistance.

The Manager Application will query the database to retrieve information regarding employee information and accounting information. The Manager Application will have an option to refresh the taskboard, which will cause the Manager Application to query the database. If a new task is created using the Manager Application, the Employee Application will be pinged, causing the Employee Application to query the database.

Section 3.2: Identifying Subsystems

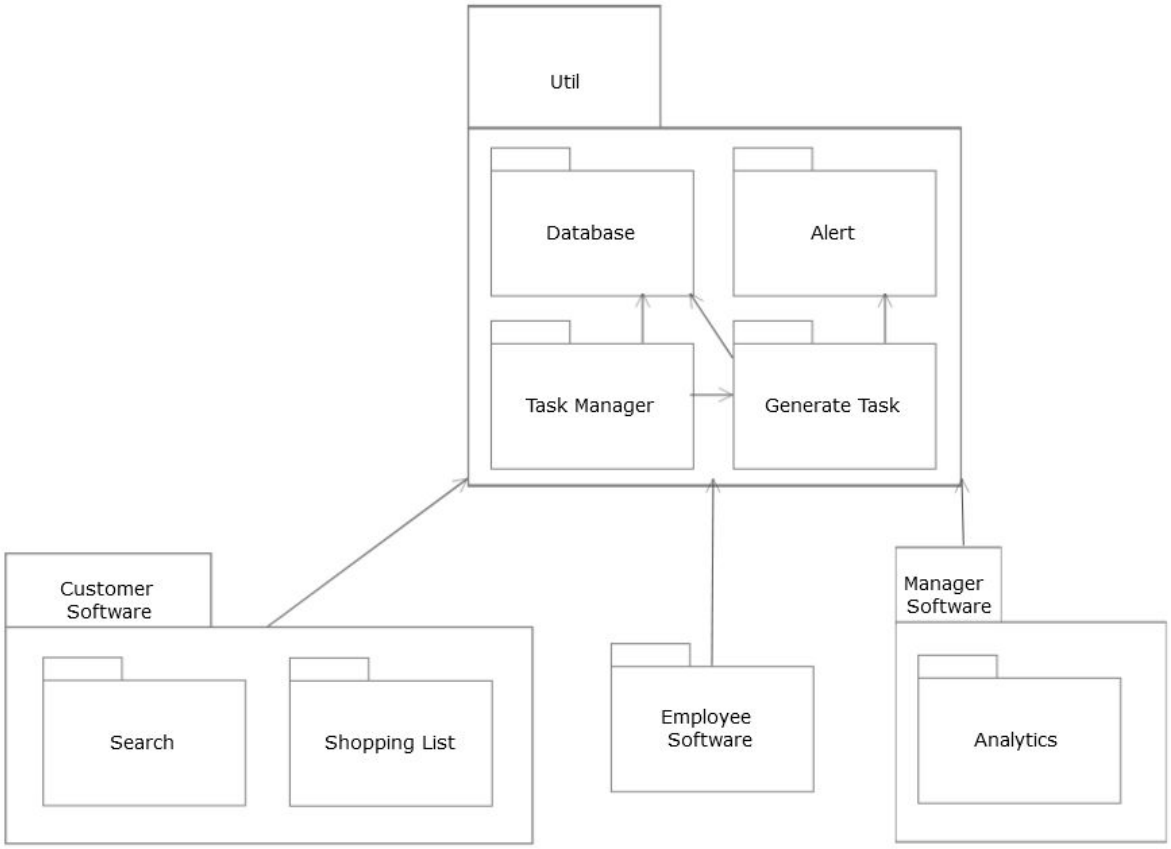


Figure 3.2-1

The util software package will control access to the database and manage the taskboard. This package interacts with other packages and contains functions that are used by multiple packages so as to reduce redundancy.

The customer software package will be able to search for items and manage a shopping list. It interacts with the util package to access the database to perform those actions. The customer software package also interacts with the util package to generate tasks, such as when a customer removes an item from their cart to be returned to the shelves or when a customer requests assistance.

The employee software package will be used to view the task board managed by the util package. When a task is generated by another package, the util software package will send an alert that will be received by the employee software package.

The manager software package will be used to both create and view tasks that are managed by the util package as well as query the database and perform analytics such as sales figures and employee task completions.

Section 3.3: Mapping Subsystems to Hardware

The system will run on multiple devices. There will be a Customer Application, an Employee Application, and a Manager Application.

The Customer Application will be run on either (1) Customer Assistance Terminals which exist around the store or (2) a customer's own smart device. Customer Assistance Terminals contain a tablet which will be running the Customer Application.

The Employee Application and the Manager Application will run on smart devices for employees and managers respectively. Grocery stores will determine whether the smart devices will be provided to employees and/or managers for usage in the store or whether the smart devices should be the personal smart devices of employees and managers.

Section 3.4: Persistent Data Storage

The system needs to save data regarding employees and stock past a single execution of the system. The system will be stored in a relational database using mySQL. For this project, the database shall be hosted on an AWS server.

items(*name*, *brand*, *type*, *RFID*, *price*)

The **items** table contains information about all products that could potentially be sold in the store. The *RFID* column is the primary key for this table since the RFID for each product is unique. The database should contain at most 4,294,967,296 tuples so as to ensure unique RFID numbers. The sample database used contains 1,000 tuples.

Column Name	Description	Type
name	This is the name of the product.	VARCHAR(65535)
brand	This is the name of the brand that produces the product.	VARCHAR(65535)
type	This is the type of product/category under which the product falls. This will be used to help identify where the product should be located in the store.	VARCHAR(65535)
RFID	This is the number represented by the RFID tag that should be attached to the product when in store.	INT UNSIGNED
price	This is the price of the product in USD.	DECIMAL(10,2)

stock(*itemRFID*, *amt*, *expirationDate*, *location*)

The **stock** table contains information about which items are in stock within the store and the backroom. There may be multiple tuples in the database with the same *itemRFID*. The first case is that the *expirationDate* of the tuples are all unique. This is to ensure that items are still shelved, but expired items will be removed. The second case is that one tuple has a *location* in the store while the other tuple has the *location* of BackRoom. This is to separate the items that are in stock in the store and the items that are in stock in the backroom.

Column Name	Description	Type
itemRFID	This is the RFID of the product.	INT UNSIGNED
amt	This is the amount of the product at *location*.	INT UNSIGNED
expirationDate	This is the expiration date of the item. This may be empty for some items, which do not expire.	DATETIME
location	This is the location of the product in the store. This includes the backroom.	VARCHAR(65535)

employees(*lastName*, *firstName*, *ID*, *role*)

The **employees** table contains basic information about employees. The *ID* is the primary key for this table since each employee's employee ID should be unique. The database should contain at most 4,294,967,296 tuples so as to ensure unique employee numbers. The sample database used contains 60 tuples.

Column Name	Description	Type
lastName	This the last name of the employee.	VARCHAR(65535)
firstName	This is the first name of the employee.	VARCHAR(65535)
ID	This is the employee's employee ID.	INT UNSIGNED
role	This is the role of the employee.	SET(Employee, Manager)

hours(*employeeID*, *day*, *checkIn*, *checkOut*, *dayHours*)

The **hours** table contains information about employees' hours per day.

Column Name	Description	Type
employeeID	This is the employee's employee ID.	INT UNSIGNED
day	This is the date that the employee checked in/out.	DATE

checkIn	This is the time when the employee checked in.	TIME
checkOut	This is the time when the employee checked out.	TIME
dayHours	This is the number of hours that the employee worked on *day*.	FLOAT

tasks(*taskName*, *description*, *state*, *employee*, *timeCreated*, *timeCompleted*)

The **tasks** table contains information about tasks that have been created for employees to complete.

Column Name	**Description**	**Type**
taskName	This is the name of the task.	VARCHAR(65535)
description	This is a description of the task.	LONGTEXT
state	This is the state of the task, describing whether the task is untaken (Incomplete), currently being completed by an employee (In Progress), or completed by an employee (Complete)	SET(Incomplete, In Progress, Complete)
employeeID	This is the employee ID of the employee who either is completing the task or has completed the task.	INT UNSIGNED
timeCreated	This is the time that the task was created.	DATETIME
timeCompleted	This is the time when the task was completed. This may be blank for some tasks.	DATETIME

sales(*transactionID*, *time*, *totalPaid*)

The **sales** table contains information about transactions that were made in the grocery store. The *transactionID* and *time* are the primary keys for this table.

Column Name	**Description**	**Type**
transactionID	This is the transaction ID of the transaction.	INT UNSIGNED
time	This is the time that the transaction was made.	DATETIME
totalPaid	This is the total amount paid by the customer.	DECIMAL(10,2)

transactions(*transactionID*, *item*)

The **transactions** table contains information about what items were bought with each transaction.

Column Name	**Description**	**Type**
transactionID	This is the transaction ID of the transaction.	INT UNSIGNED

time	This is the time that the transaction was made.	DATETIME
item	This is the RFID of the item that was bought in the transaction.	INT UNSIGNED
amt	This is the amount of the item that was bought in the transaction.	INT UNSIGNED

logins(*username*, *password*, *accountType*)

The **logins** table contains information about login information for usage of the Customer Application, the Employee Application, and the Manager Application on smart devices. The sample database used contains 70 tuples.

Column Name	**Description**	**Type**
username	This is the username of the account.	VARCHAR(65535)
password	This is the password that is used to log into the account.	VARCHAR(65535)
accountType	This is the type of the account, which will determine which application the user can access.	SET(Employee, Manager, Customer)

shopping-lists(*username*, *item*)

The **logins** table contains information about customers' shopping lists.

Column Name	**Description**	**Type**
username	This is the username of the account.	VARCHAR(65535)
item	This is the RFID of the product in the customer's shopping list.	INT UNSIGNED

Section 3.5: Network Protocol

The system shall use the Python mySQL connector. This choice was made because (1) a mySQL database will be used, and (2) the system will be coded in Python.

Section 3.6: Global Control Flow

The system is typically event-driven. Events are typically initiated by a customer or manager. Customers may initiate events to search for an item, to check the price of an item, to remove an item from their cart, or to purchase items; these will interact with only the database. Customers may also initiate events to request for help, which will interact with both the database and the Employee Application. Managers can create tasks, which will interact with both the database and the Employee Application. Rather than specifically waiting for an event, the events directed towards the Employee Application will work similarly to interrupts; in other words, the events are asynchronous.

The system is of event-response type, notably with regards to pinging the Employee Application for a notification of a new task. Those events may be generated by customers or managers.

Section 3.7: Hardware Requirements

The system depends on screen display on tablets and smart devices, RFID sensors, an Internet connection, and either (1) database disk storage on AWS or (2) an on-site database disk storage and a local network connection. The screen display will display the Customer Application, the Employee Application, and the Manager Application. The RFID sensors are used to identify items for purchase, price checking, or returns.

The database disk storage on AWS will be used to store persistent data; grocery stores may opt to have a backup disk storage copy of the database available on site in case of Internet outage. As the database exists on AWS, an Internet connection will be required to access it. If an on-site database disk storage is used in lieu of an AWS database, then an Internet connection is still necessary for customers to access and update their shopping list, and a local network connection could be necessary to access the on-site database.

Section 4: Data Structures

The data structures used in the program are simple because the backbone of the program is a database. The main data structure used will be a linked list for the representation of the shopping cart and another linked list for the task list. A linked list is best suited for this task so items and tasks can be removed out of order. The main criteria used is flexibility and not performance because our program is not doing enough computations to require performance considerations.

Section 5: User Interface Design and Implementation

Some minor changes were made from the initial screen mockups that were developed for Report #1. When the screen mockups, processes, and responses were first developed, “ease-of-use” into account was not taken into account as much as was done with the later designs. Once the number of clicks it would take for the customer/employees to access a certain process was determined, it was discovered that there were some extraneous steps in the processes.

The current version of our Returns Terminal in Figure 5.1 as shown below will be examined:

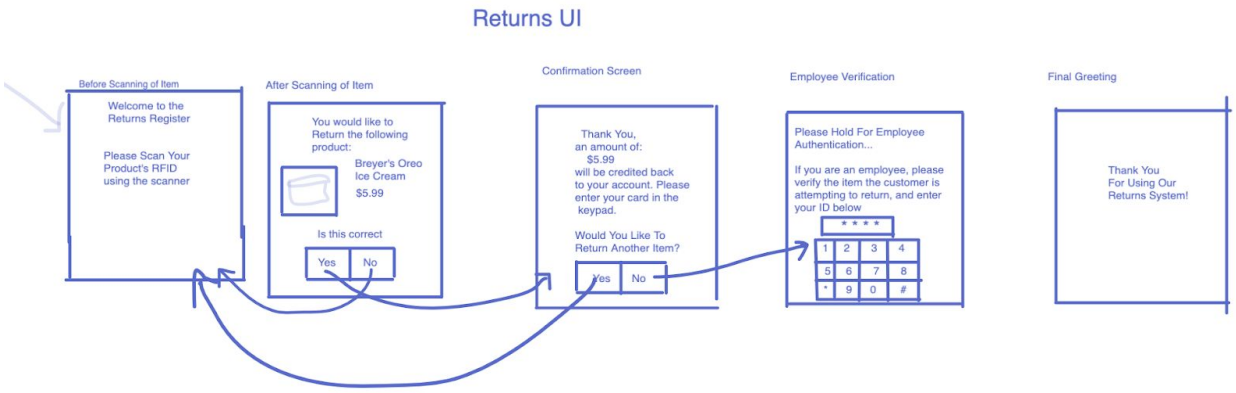


Figure 5-1

When the processes for the Returns system was initially drawn up, the employee verification screen came before the confirmation screen, and the employee would perform the confirmation. However, using a test case where the customer has to perform multiple returns, it was quickly discovered that the employee would need to enter in his or her verification ID each time for every single item. Due to this, confirmation access was given to the user who selects whether or not they want to return another item, and once the user has returned all of their items, the employee can then enter in their verification ID once for all of the items as a summed return. This drastically reduced the number of clicks and proved to be a much more efficient process.

Apart from the returns terminal, ease-of-use was kept in mind while designing our other terminals, and that the initial version still proves to be the most efficient way of completing the required processes. Overall, all of the terminals were kept simple and provided a lot of detail along with limited keystrokes to the user to develop a user-friendly, guided system for the supermarket.

Section 6: Design of Tests

Test-case Identifier: TC-1 **Use-case Identifier:** UC-8 **Pass/Fail Criteria:** The test passes if the customer checkout is completed without any errors returned to the customer **Input Data:** Item barcodes, Customer Payment information, Remove item button	
Test Procedure: Step 1: Have the customer scan items, remove an item in the process, and then proceed to checkout.	**Expected Result:** For Step 1: The scanned items will all be in a list with their price and the calculator will return the total with the state tax. If the customer wishes to return an item, then the item will be removed from the cart, and a new total will be calculated. The customer's payment method will be asked and the process will be completed once the customer pays the balance.

Test-case Identifier: TC-2 **Use-case Identifier:** UC-2 **Pass/Fail Criteria:** The test passes if the item is located and the location of the item in the store is returned to the user. If the item is invalid, that information is also returned to the user. **Input Data:** Item's unique identifiers such as barcode, name, etc.	
Test Procedure: Step 1: Enter the barcode ID for an item that is in the store. Step 2: Enter a random barcode ID for an item that is not in the store (an invalid item)	**Expected Result:** For Step 1: The location of the item is presented to the employee/customer in a way that makes it easy for the user to locate in the store. For Step 2: The screen displays to the user that the item is invalid.

Test-case Identifier: TC-3 **Use-case Identifier:** UC-13 **Pass/Fail Criteria:** The test passes if the user goes through the entire returns process without returning any errors in the system. **Input Data:** Return item barcode, customer repayment information, Employee ID

Test Procedure: Step 1: Have the customer return 1 item that is valid and have them go through the returns terminal. Step 2: Have the customer return an empty box that an employee identifies and uses the terminal to indicate this.	**Expected Result:** For Step 1: The customer goes through the returns terminal and returns the items that they want to. The employee verifies their returned items and enters in their Employee ID, and the employee will get the refund through their requested payment method. For Step 2: Same steps as Step 1, however, the employee will check the items and enter that item is invalid, and enter this in the Returns terminal when asked to enter their employee ID, and this will quit the Returns process.

Test-case Identifier: TC-4 **Use-case Identifier:** UC-15 **Pass/Fail Criteria:** The test passes if a valid employee ID is entered, and the correct working schedule is returned for the employee along with the employee wage. Using the schedule and wage, the system should calculate the employee salary. **Input Data:** Employee ID	
Test Procedure: Step 1: Enter a valid Employee ID in the terminal, then select calculate salary. Step 2: Enter an invalid Employee ID	**Expected Result:** For Step 1: This returns the employee's schedule initially, and when asked to calculate salary, the terminal uses the hours of the employee and the employee wage to calculate weekly salary and that amount is returned and displayed. For Step 2: This returns that this is not a valid employee ID, and refreshes to the home screen automatically in ~5 seconds.

Test Coverage Strategy

In terms of coverage, the code is being tested to the greatest degree through TC-1 and TC-3. These are the two terminals that customers will be interacting with when checking out and returning their items. To ensure sure that this process is flawless and contains no bugs, these designed test cases examine the majority of scenarios that would occur when customers are using this terminal.

Through TC-1, the checkout scenario is analyzed and tested using various items to check that adding a large amount of items does not slow down the system; if our program passes this test, then the customers should be able to buy a large sum of items with no problems. The process behind customers wanting to put an item back during checkout is also tested, and if the system passses this as well, one can be confident that the system can handle most customer checkouts.

Through TC-3, the returns process is analyzed and tested much like the checkout process was tested, but with an added step: employee verification. Much like checkout, it must be ensured that returning a large sum of items does not slow down the system or cause bugs, especially since the returns terminal is responsible for refunding the customer as well. There is also a test to make sure that the employee verification is working well, and through this, most if not all possible return scenarios are tested.

Integration Testing Strategy

The integral part of the program is the connection between the user and the database. Because this project has few systems, a Big Bang approach to integration testing should be sufficient. In order for the system to work every part would need to be there. As the user will be interacting with a GUI ,there needs to be a proper interaction between them. The program will have helper methods called by the GUI in order to call the database, and then the database will send the proper information back to the GUI so that it can be displayed. To test these functions, the database will be populated with sample values manually, and then the GUI will be used to access this data through the search function.

The actual testing will involve a check to make sure the GUI is actually calling the methods, followed by a check to make sure the methods are properly referencing the database, which finally is followed by checking to make sure the data is being retrieved by the database and implemented by the GUI.

For example, the integration of test case 1 would be tested by connecting all the parts together and then if there are errors using test markers in order to check if the program is properly contacting the database or not. In this specific case, that would be testing that the GUI is properly calling the method to remove an item and that the item is being removed from the cart.

Section 7: Project Management and Plan of Work

Section 7.1: Merging the Contributions from Individual Team Members

The work was divided such that each team would typically be responsible for one section of a report. There were difficulties in getting members to start work earlier and in contacting certain members. Depending on whether certain members did their part, points were reassigned based on what work was done by each member, and future parts were reassigned based on who needed to do more.

As each team had different writing styles, ensuring consistency was rather difficult. Team A would typically look over sections in order to ensure consistency among the different writing styles to the best of their ability. This was difficult at times because not all members would complete their work early, meaning that very little time was available for reading and editing.

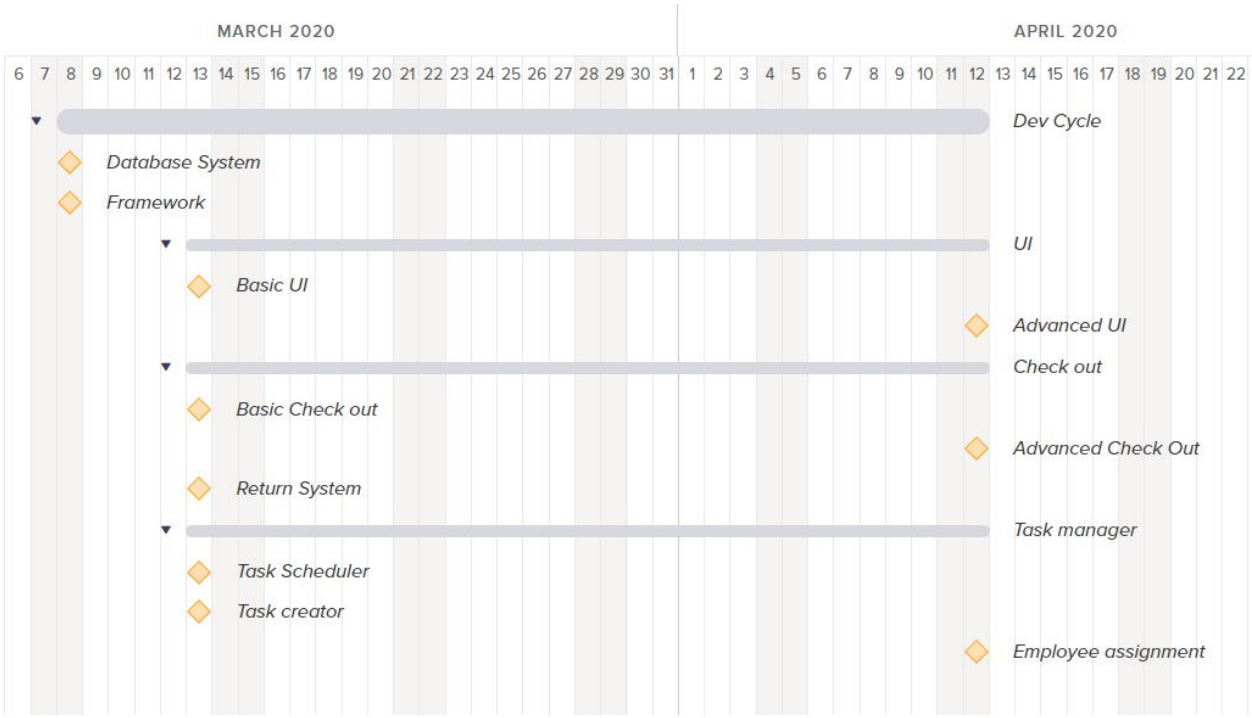
Uniform formatting and appearance was ensured by having a "skeleton" of the report available on Google Docs. A Google Doc for each part of the report was made, and headings for each section(s) and subsections of that report were placed alongside the instructor's description of the section to create a "skeleton" of the report. Before submission of the report, all of the instructor's descriptions of the sections were removed so as to minimize the amount of unnecessary content in the report.

Section 7.2: Project Coordination and Progress Report

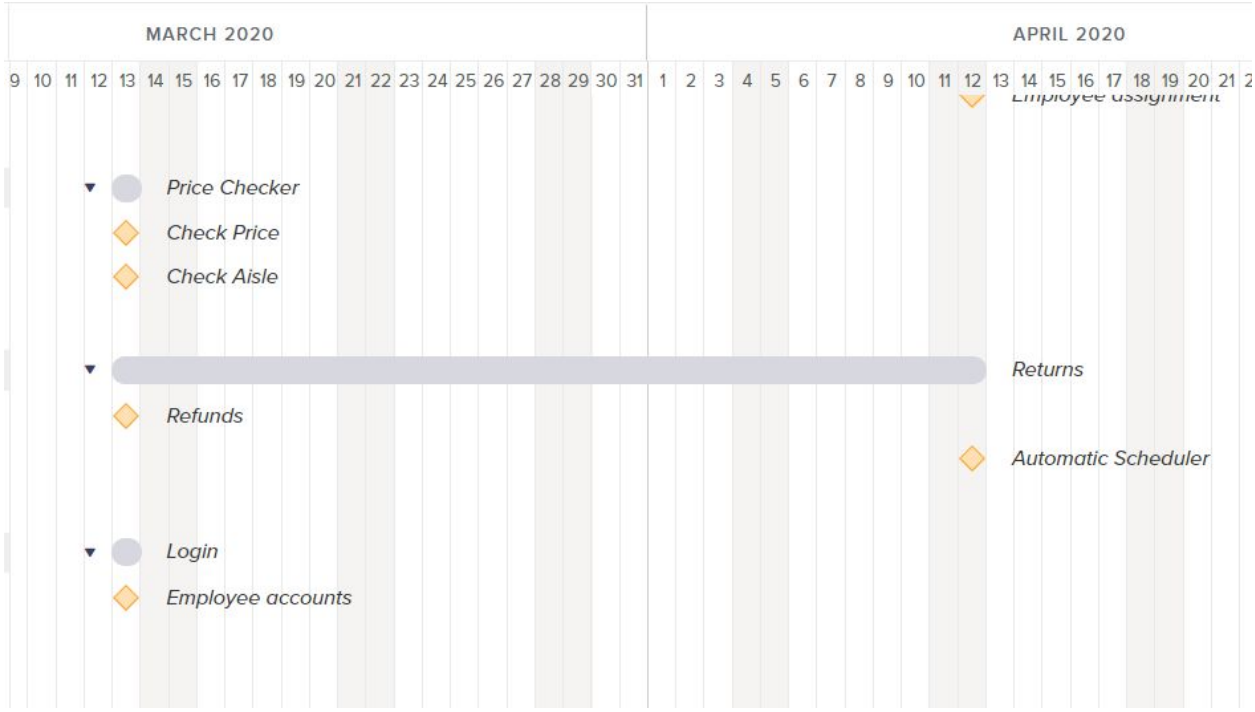
Currently the database and utility functions for the database are being worked on. Basic functions of the UI are also being worked on. Task management functions are almost complete. The database has been partially populated with data.

The team manager, Andrew Saengtawesin, is keeping track of due dates and tasking group members as necessary.

Section 7.3: Plan of Work



Section 7.3-1



Section 7.3-2

Section 7.4: Breakdown of Responsibilities

Responsibility	Group Responsible
Database	Group A
Task Manager	Group A
Check Out	Group B
Returns	Group B
Price Checker	Group C
Item Locator	Group C
Login	Group C

Group A	Andrew Saengtawesin
	Kimberly Chang
Group B	Jake Totland
	Andrew Vincent
Group C	Amali Delauney
	Abhinandan Vellanki
	Harsh Desai

Group A will be responsible for coordinating integration and for integration testing.

Section 8: References

[1] Wikipedia Contributors (2019). *Software architecture*. Wikipedia.
https://en.wikipedia.org/wiki/Software_architecture
[2] Wikipedia Contributors (2019). *Blackboard system*. Wikipedia.
https://en.wikipedia.org/wiki/Blackboard_system
[3] Wikipedia Contributors (2019). *Event-driven architecture*. Wikipedia.
https://en.wikipedia.org/wiki/Event-driven_architecture
[4] Wikipedia Contributors (2019). *Peer-to-peer*. Wikipedia.
https://en.wikipedia.org/wiki/Peer-to-peer

References 1 through 4 are used for Section 3.1 to assist in identifying the architectural styles used in this project.